# Pharo's Tips and Tricks

Stéphane Ducasse

Square Bracket tutorials
October 28, 2017
master @ d69dc78*

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

# Contents

# Illustrations

This little booklet contains some tips and tricks to improve and custom your Pharo experience. If you have tips and tricks please share them with us.

CHAPTER 1

# Startup Preferences

Have ever been wanted to always do the same actions before starting to work with a new image? Since its version 2.0, Pharo supports startup preferences. You can specify actions that are automatically executed when you launch the system. Startup preferences were implemented by B. van Ryseghem and M. Martinez-Peck.

This chapter is an update for Pharo 6.0 of a blog post of Martinez-Peck that covered this topic. Before showing how you can take advantage of startup preferences, we will look at the old way to execute automated actions at startup. Then we present startup preferences.

## 1.1 Traditional startup execution

Pharo supports the execution of specific expressions at startup time. Traditionally there have been several ways to specify them: (1) provide a script file or (2) an expression as command-line arguments, and (3) register to the startup list some actions.

### Provide a script file to the VM

Imagine that you have a file `logo.st` containing the following expressions that grab a png and display it.

```
more logo.st
>>>
(ZnEasy getPng: 'http://pharo.org/files/pharo.png') asMorph openInWindow
```

To pass a `.st` file as parameter to the VM, that is, you execute the VM like this:

```
./pharo Pharo.image st logo.st
```

You specify your image, the `st` command, and the name of the file you want to execute.

### Execute an expression passed on the command line

Now you can also execute an expression as follow:

```
./pharo Pharo.image eval "(ZnEasy getPng: 'http://pharo.org/files/pharo.png') asMorph
    openInWindow"
```

Note the use of " to surround the expression.

Remember that you can get the list of active command-line handlers as follows:

```
./pharo Pharo.image --list
Currently installed Command Line Handlers:
    Fuel                       Loads fuel files
    config                     Install and inspect Metacello Configurations from the
    command line
    save                       Rename the image and changes file
    update                     Load updates
    printVersion               Print image version
    st                         Loads and executes .st source files
    test                       A command line test runner
    clean                      Run image cleanup
    get                        Install catalog projects from the command line (consult
    catalog at http://catalog.pharo.org)
```

What we did, you execute the VM, pass a Pharo image as parameter and then the file. This will be executed during system startup. What's the problem with this? If you always want to execute the startup in different images you always need to open your image this way and from command line. Moreover, this file is hand-coded and not versioned in Monticello.

### Using the start up list

Pharo infrastructure offers the possibility for classes to register themselves to a startup list. The system ensures that the `startUp` and `shutDown` methods of the registered classes are executed during the start up and shut down of the system.

Therefore you can register your own class in the system startup list and implement `startUp` message to do whatever you want to do. The problem is that your class is not present in the distributed images of Pharo. Therefore you need to manually first load my own code and this is not optimal. In addition, it is better to keep the startUp list used for system level operations.

Now since Pharo2.0 there is a startup loader that we will present next.

## 1.2 Startup Preferences

The `StartupLoader` class searches for and executes `.st` files from certain locations that we will explain in details below. On mac it looks into your library preferences folder.

```
Library/Preferences/pharo/mystartup.st
Library/Preferences/pharo/5.0/mystartupFor50only.st
Library/Preferences/pharo/6.0/mystartupFor60only.st
```

As you see you can structure your preferences depending on the version to which they should be applied.

## 1.3 Locations

To know the locations that are visited to find start up files, you can execute the following expression:

```
StartupLoader preferencesGeneralFolder
>>>
File @ /Users/ducasse/Library/Preferences/pharo
```

```
StartupLoader preferencesVersionFolder
>>>
File @ /Users/ducasse/Library/Preferences/pharo/7.0
```

The message `preferencesGeneralFolder` returns the location for the startup scripts common to all Pharo versions. The message `preferencesVersionFolder` returns the location for the startup scripts specific to the version of the current image.

The location search order from the most general to the most specific are the following ones: general, version specific or image folder specific. As you can see the order is from the most general to the most specific. Moreover, it does not stop when it finds files in any of the locations, they are all searched and executed. More specific scripts can even override actions set in more general ones.

### General preference folder

General means that it applied to all Pharo versions. This folder contains startup script for all the images you will open.

- On OSX and Pharo, it is `Library/Preferences/pharo/`.
- On linux, it is in your home directory in the `.config` folder, e.g., `/home/YOUR/.config/pharo`.
- On windows, it is

```
C:\Users\Name_Of_Your_User\AppData\Roaming\Pharo
%userprofile%\AppData\Roaming\Pharo
```

In this place, `StartupLoader` will load **all** existing `.st` files. This type of startup is useful when we have something to execute for all images of all Pharo versions.

### Preference version folder

Now if you want to only get some actions executed for a specific version, define a folder with the version number inside the general preference folder. On Mac OS X, a specific 6.0 folder is `Library/Preferences/pharo/6.0/'` in your user account. This type of startup is useful when we have something to execute for all images of a specific Pharo version.

### The image folder

The startup only searches for a file called `'startup.st'`. So if you have such a file in the same directory where the image is, then such script will be executed automatically. This type of startup is usually used to do something that is application-specific or something that only makes sense for the specific image you are using. Now you might ask why we don't search the image folder for multiple .st files. This is because it is normal for the image folder to contain `.st` files unrelated to preferences - such as from any file out. Using one specific file `'startup.st'` avoids problems. Be careful if you already were sending your own `'startup.st'` as parameter to the VM because it will be executed twice.

You can query whether the `StartupLoader` is enabled using the messages `allowStartupScript` and you can change its status using `allowStartupScript:` sent to the class `StartupLoader`.

So you know where the system will search startup files. Now you cannot just write any script. You should use StartUp Actions as we will explain next.

## 1.4 Startup Actions

Directly putting code in the files is easy, however, it is not the best choice. For example, what happens if there are certain scripts you want to execute only once on a certain image but some other code that you want to execute each time the image starts? To solve that, among other things, StartupLoader reifies actions into objects of the class `StartupAction`. Let's see how we use them:

```
| items |
items := OrderedCollection new.
items add: (StartupAction
            name: 'Basic settings'
            code: [
                Author fullName: 'YourName'.
                GTGenericStackDebugger alwaysOpenFullDebugger: true ]).
```

```
StartupLoader default
    addAtStartupInPreferenceVersionFolder: items
    named: 'basicSettings.st'.
```

First we create an instance of `StartupAction` using the message `name:code:`. We pass as argument a name and a block that we want to run. Note that you can also pass a string as parameter of the `code:` message part but this is less readable.

In this example, we just set the author identification to 'YourName' and we put a setting to always open the debugger (no pre-debugger window). So far nothing special.

The magic comes with the last line, the message `addAtStartupInPreferenceVersionFolder: items named: aFileName` receives a list of startup actions and a filename and stores the actions in a file with the passed argument. So in this case we have only one action called `'Basic Settings'` and it will be saved in a file called `'basicSettings.st'`. But where?

Since we used the message `addAtStartupInPreferenceVersionFolder:named:` (notice the `InPreferenceVersionFolder`), it will be created in the following place `pharo/pharoVersionNumber/`. The message `addAtStartupInGeneralPreferenceFolder:named:` stores files in the general folder (`pharo`) and the `addAtStartupInImageDirectory:` stores besides the image.

Notice that with the first two messages we can specify the file name but with the last one we can't. Remember the last one is always called `'startup.st'`. If you are lazy and don't want to think the name yourself, you can just use `addAtStartupInPreferenceVersionFolder:` which creates a file called `'startupPharoNN.st'` or `addAtStartupInGeneralPreferenceFolder:` that creates a file named `'startupPharo.st'`.

## 1.5 StartupLoader

We saw that when executing the message `addAtStartupInPreferenceVersionFolder:named:` or any of its variant, a file is created with the code we want to evaluate. Then, when the system starts it will find our file and execute our code. But, how is the resulting file? Exactly as the code we provided? No! Look how our example file is generated. You can find it using the following expression: `StartupLoader preferencesVersionFolder / 'basicSettings.st'`

You should get a result similar to the following one:

```
StartupLoader default executeAtomicItems: {
        StartupAction
      name: 'Basic settings'
      code: [ Author fullName: 'YourName'.
      GTGenericStackDebugger alwaysOpenFullDebugger: true ].
}.
```

So as you can see, the file is generated by sending a collection of actions to `StartupLoader default executeAtomicItems:`. In this example the collection has only one action, but it would have more if our example had more. So now the StartupLoader will execute all the actions found in the file.

## 1.6 Controlling action execution and errors

Once your image is built and you save it, it can be annoying that the actions are systematically executed. This is why you can specify that an action should be executed only once or at each image startup.

Sending the message `name: nameOfItem code: code runOnce: aBoolean` to an action, you make sure that if an action has already been executed in the current image, it will not be executed again. Note that executed actions are stored in the singleton instance of the class `StartupLoader`.

If an action generates an error, the action is not registered as executed. In addition, errors are also stored in the `StartupLoader` singleton so you can query them after starting the image by inspecting the result of `StartupLoader default errors`.

## 1.7 Testing your startup scripts

To try out your startup scripts you can simply to do the following: create them, add them to the place of your choices and send the message `loadStartupScript` the `StartupLoader` singleton. The following example shows this process.

```
| item1 item2 |
item1 := StartupAction
  name: 'Open Help'
  code: [ Workspace openContents: 'Here is just an example of how to use the
    StartupLoader.
  It should only be displayed once. You can also see StartupLoader class>>#example'
    label: 'Help']
  runOnce: true.
item2 := StartupAction
  name: 'Open Workspace'
  code:  [ Workspace openContents: 'I should be displayed each time' ].
StartupLoader default
  addAtStartupInGeneralPreferenceFolder: {item1. item2}.
StartupLoader default loadStartupScript.
```

## 1.8 Some nice configurations

Here are some nice expressions that you can use in a startup preferences. Have also a look at the Setting Browser to find others.

### Debugger

You can set the debugger to never open the preview and to filter common message sends.

```
GTGenericStackDebugger
  alwaysOpenFullDebugger: true
```

### Changing fonts

You can change the fonts.

```
StandardFonts defaultFont: (LogicalFont familyName: 'Consolas' pointSize: 10).
GraphicFontSettings resetAllFontToDefault.
```

### Changing desktop

You can change your desktop color and remove the Pharo logo.

```
PolymorphSystemSettings
  desktopColor: Color gray;
  showDesktopLogo: false.
UITheme currentSettings fastDragging: true.
```

### Changing syntax style

You can customize the text editor as well as the style of the syntax highlighter.

```
TextEditorDialogWindow autoAccept: true.
SHPreferences setStyleTableNamed: 'Solarized'.
```

### Freetype

You can enable the Free Type fonts.

```
FreeTypeSystemSettings loadFt2Library: true.
```

### Sharing

You can specialize Monticello to share your package cache.

```
| sharedPackageCacheDirectory |
sharedPackageCacheDirectory := '/Users/ducasse/Pharo/localRepo/' asFileReference
    ensureCreateDirectory.
MCCacheRepository default directory: sharedPackageCacheDirectory.
MCDirectoryRepository defaultDirectoryName: '/Users/ducasse/Pharo/localRepo/'.
```

### Password

Setting the user and password of some Monticello repositories

```
(MCRepositoryGroup default repositories
  select: [:each |
    (each isKindOf: MCHttpRepository)
      and: [((each locationWithTrailingSlash includesSubString: 'www.squeaksource.com')
          or: [each locationWithTrailingSlash includesSubString:
    'http://ss3.gemstone.com/ss/'])]
      ])
   do: [ :each |
      each
        user: 'YYY';
        password: (FileLocator home / 'Mypass' /'mcrepositoriesPassword.txt') contents ]]
```

### Password

You can also add conditional actions based on the fact that the image is running in headless mode or not.

```
Smalltalk isHeadless ifFalse: [
  StartupLoader default executeAtomicItems: {
    StartupAction name: 'Image Setup' code: [ ... ]
    }
  ]
```

### Github

You can also customise the git integration.

- First we enable the integration of metacello.
- Then we identify the rsa key (which should be identical to the one placed in your github account).
- We set the github account and associated password.
- We also set support for sharing clone repositories between multiple images. This is to avoid to have to clone multiple times the same code base. `sharedRepositoriesLocationString:` specifies the location where all your clone repositories will be located.

- Finally we set the system repositories (note that this feature of Iceberg is about to disappear) and for this we set the location of Pharo cloned code. You may have to supply a subdirectory such as 'src' using the message `subdirectory:`.

```
StartupPreferencesLoader default executeAtomicItems: {
  StartupAction
    name: 'Git Settings'
    code: [
      FileStream stdout cr; nextPutAll: 'Setting the ssh credentials'; cr.

      Iceberg enableMetacelloIntegration: true.

      IceCredentialsProvider useCustomSsh: true.
      IceCredentialsProvider sshCredentials
          username: 'git';
          publicKey: '/Users/YOUR/.ssh/id_rsa.pub';
          privateKey: '/Users/YOUR/.ssh/id_rsa'.

      IceCredentialsProvider
        plaintextCredentials: (IcePlaintextCredentials new
                username: 'YOUR';
                password: 'YOURGITHUBPASS' ; yourself ).

      IceRepository
          shareRepositoriesBetweenImages: true;
          sharedRepositoriesLocationString: '/.../Pharo/PharoCodeBase/'.

      Iceberg showSystemRepositories: true.
      IceRepository registry
        detect: [ :each | each name = 'pharo' ]
        ifFound: [ :repo |
          repo location: '/.../Pharo/PharoCodeBase/pharo' asFileReference ].

      FileStream stdout
        cr; nextPutAll: 'Finished'; cr ].
}.
```

## 1.9 How to split your settings in files and actions?

So you may have noticed that:

1. `addAtStartupInPreferenceVersionFolder:` and friends expect a list of actions;
2. you can have multiples files. How do you split your code? There is basically no restriction from the design of StartupLoader.

You can have as many actions per files and as many files as you wish. An action has a block of closure that can contain as much code as you want.

We found that one way of splitting your code is when some actions need to be executed only once and some other each time. Another reason may be some code which may be expected to fail for some reason. If it fails, the code after the line that generated the error won't be executed. Hence, you may want to split that code to a separate action.

## 1.10 Be careful with the cache!

To support the `runOnce:` property, actions are stored in the singleton instance of StartupLoader. After an action is executed (if executed correctly), the action is stored and marked as executed. It may happen that later on you modify the scripts by hand, or you change the rules and re-store them or some kind of black magic. If you change some of these, we recommend to do 2 things:

- Remove all existing files from the preference directories (no script is removed automatically). Check methods `remove*` in `StartupLoader`.
- Remove the existing stored actions in `StartupLoader`. Check the method `cleanSavedActionsAndErrors` and `removeAllScriptsAndCleanSavedActions`.

## 1.11 Conclusion

This tool is really an excellent companion to improve your productivity and build automatically working environments where you feel at home. So use it and share your nice settings with the community.